Functional Array Programming for Cluster Architectures

Clemens Grelck
University of Amsterdam
C.Grelck@uva.nl

Thomas Macht
VU University Amsterdam
T.Macht@student.vu.nl

1. Introduction

Single Assignment C [5] is a functional data parallel language specialising in array programming. SAC combines high productivity programming with efficient parallel execution. Data parallelism in SAC is based on array comprehensions in the form of `with`-loops that are used to create immutable, truly multi-dimensional array values or to perform reduction operations.

The SAC compiler goes to great lengths combining and optimising `with`-loop-based intermediate code [4]. In doing so SAC compiler systematically transforms programs from a human-friendly representation to one that is amenable to efficient machine execution. Eventually the SAC compiler emits platform-specific ISO C compliant code. Currently, the compiler includes backends for symmetric multi-core multi-processor systems with shared address space architectures [2], GPUs (based on CUDA) [6], the Micro-Grid many-core architecture [3] as well as heterogeneous systems of multi-core general-purpose processors and GPUs [1].

A clear gap in the SAC compiler's portfolio of target platforms are cluster-like architectures with distributed address spaces. The goal of our current work is to close this gap and add efficient cluster support to SAC compiler and runtime system.

2. Approach

Adhering to the design and philosophy of SAC we aim for a completely automatic solution that maps entirely system-agnostic programs to cluster architectures and, nonetheless, achieves competitive performance for the compute-intensive applications that SAC targets. Since the programming language SAC supports arbitrary index functions on arrays, we cannot expect to be able to fully analyse array access patterns, at least not in the general case. Therefore, we follow a distributed shared memory (DSM) strategy. While DSMs historically have a reputation for poor performance, it is now time to re-think this approach as technology has thoroughly shifted the relation between network and memory speed. Correspondingly, interest in Software DSMs is again on the rise [8].

3. Why a Software DSM solution?

Distributed shared memory provides a shared memory abstraction on top of a physically distributed memory. An overview of issues of DSM systems can be found in [7]. These early DSM systems have not been adopted on a large scale due to shortcomings in performance. Explicit message passing, and in particular MPI, remain the predominant programming model for clusters.

However, Ramesh et al. suggest that it is time to revisit DSM systems [8]. They argue that early DSM systems were not successful because of slow network connections at the time. Today, network bandwidth is comparable to main memory bandwidth, and network latency is only one order of magnitude worse than main memory latency. These developments pretty much reduce Software DSMs to a cache management problem.

4. Why a custom DSM system?

In order to support distributed memory systems, we could run a SAC program on top of an existing Software DSM system. Instead, we decided to integrate a custom DSM system into the SAC compiler and runtime system. This allows us to exploit SAC's functional semantics and the very controlled parallel execution model of `with`-loops. Since variables in `sac` have write-once/read-only semantics, we do not have to take into account that they could change their value. Furthermore, parallelism only occurs in `with`-loops and while arbitrary variables can be read in the body `with`-loop, only a single variable is written to.

5. Implementation and evaluation

During the presentation we outline the major design choices made in our implementation of a custom Software DSM for functional array processing as part of the SAC runtime system as well as the major extensions of the SAC compiler to make use of it. We further report on our experimental findings involving a variety of benchmarks and draw conclusions regarding the overall approach.

References

[1] M. Diogo and C. Grelck. Towards heterogeneous computing without heterogeneous programming. In *Trends in Functional Programming, 13th Symposium, TFP 2012, St.Andrews, UK*, LNCS 7829, pages 279–294. Springer, 2013.

[2] Clemens Grelck. Shared memory multiprocessor support for functional array processing in SAC. *Journal of Functional Programming*, 15(3):353–401, 2005.

[3] Clemens Grelck, Stephan Herhut, Chris Jesshope, Carl Joslin, Mike Lankamp, Sven-Bodo Scholz, and Alex Shafarenko. Compiling the Functional Data-Parallel Language SAC for Microgrids of Self-Adaptive Virtual Processors. In *14th Workshop on Compilers for Parallel Computing (CPC'09), IBM Research Center, Zürich, Switzerland*, 2009.

[4] Clemens Grelck and Sven-Bodo Scholz. Merging compositions of array skeletons in SAC. *Journal of Parallel Computing*, 32(7+8):507–522, 2006.

[5] Clemens Grelck and Sven-Bodo Scholz. SAC: A functional array language for efficient multithreaded execution. *International Journal of Parallel Programming*, 34(4):383–427, 2006.

[6] Jing Guo, Jeyarajan Thiyagalingam, and Sven-Bodo Scholz. Breaking the GPU programming barrier with the auto-parallelising SAC compiler. In *Declarative Aspects of Multicore Programming (DAMP'11)*, pages 15–24. ACM, 2011.

[7] Bill Nitzberg and Virginia Lo. Distributed shared memory: A survey of issues and algorithms. *Distributed Shared Memory-Concepts and Systems*, pages 42–50, 1991.

[8] B. Ramesh, C. J. Ribbens, and S. Varadarajan. Is it time to rethink distributed shared memory systems? In *Parallel and Distributed Systems (ICPADS'11)* pages 212–219, 2011. ID: 1.